A Reconfigurable Hardware Application for Machining of Metal Parts

William Cash,
Stephen Schnelle,
Sammy Lee,
Jan Frigo,
Paul Graham,
Matt Bement
Los Alamos National Laboratory

Abstract

Image processing algorithms can be used to mitigate and detect anomalies or to monitor a process such as machining of metal parts. For certain applications, typically involving high-precision components, it is desirable to detect the variability in material properties such as grain size or hardness of the workpiece material. The hardness of a finished part is determined using a high speed video camera (4000 frames/sec) and an image processing algorithm. The algorithm has a feed-forward data flow that consists of three stages: edge detection of the frame, interframe differencing to eliminate any static particles, and a cross-correlation routine to determine the chip velocity. This velocity yields information pertaining to the hardness of the metal, i.e. slower velocities correspond to softer materials, etc. Processing in real-time is necessary because of the high frame rate of the camera. Thus, a specialized hardware system is investigated. The target hardware is a specialized reduced instruction set computer (RISC) processor, the Stretch S5000, that contains a hardware accelerator built into the processor called the Instruction Set Extension Fabric (ISEF). Performance is reported in terms of algorithm run-time speed, power usage, memory implementation issues and ease of development.

I. Introduction

Machining of metal parts is generally an automated process. Current methods of detecting defects in manufactured parts involve a destructive analysis that does not provide a real-time inspection. Before using computational analysis, the measurement process involved an experienced and skillful operator to make accurate measurements. Although manual inspection may detect major defects, it cannot determine material properties such as hardness without extensive scrutiny under a microscope. Currently, automated visual methods provide greater accuracy, but involve damaging the material to conduct measurements. Therefore, a means of measuring material hardness without compromising the integrity of the material is desirable.

Work at Los Alamos National Laboratory (LANL) has shown a direct correlation between chip velocity from a metal being machined on a lathe and its material hardness. Harder metals have cutting chips that exit the part faster than those of softer metals for identical cutting parameters. Hence, the machining process can potentially be used as a continuous, nondestructive material hardness test. The unaided eye is incapable of observing this phenomenon, because of the rates at which machining processes occur. Therefore, a high-speed camera that takes 4000 frames/sec (fps) with a resolution of 110x100 pixels recorded the cutting of the part. The fine pixel resolution is used to detect features on the chip and the rapid procession of images is required to track the movement of these features.

When observing a lathe, the environment is not always constant. Signal processing algorithms offer solutions for handling the environmental variability and extracting the desired material properties from the data. A high-pass filter is implemented to remove these variances. During the machining, the lighting can be changed, or objects can obstruct the camera's view of the material. Chips from the metal can leave the lathe at various angles and affect the image. Additionally, if coolant is used, the coolant could splash and alter the image. First, a high-pass filter sharpens the edges, then interframe differencing subtracts the current frame from the previous frame to remove any spurious particles that are static between frames. This eliminates any buildup around the tool or on the camera lens. Finally, a cross-correlation algorithm extracts the velocity of the chip. Unfortunately, for even a single image the large number of computations required can prohibit real-time implementations on most conventional processors such as a Pentium, etc.

If real-time analysis is not possible, storing the large quantity of data generated by the high-speed camera can be burdensome. Even at the typical 8-bit resolution, over 10KB of memory is required to store a 110x100 pixel image. With the camera taking 4000 frames/second (fps), almost 42MB is required just to store one second worth of data. Such large amounts of data may not be thoroughly analyzed and can fill too much memory. Post-processing of the data also poses an issue as it does not allow for real-time detection and possible correction of defects. Furthermore, data transfer of 42 MB/sec alone can slow down the defect detection considerably. Hence, real-time data analysis is necessary.

Processing large amounts of data can be lengthy, even on a high-speed PC processor. A Coarse-Grained Reconfigurable Array (CGRA) hardware implementation is developed to analyze these data. Although both the Field Programmable Gate Array (FPGA) and CGRA may run slower than the traditional processor, they have the potential to outperfrom the PC processor through the use of pipelining instructions and parallelizing operations to execute in one cycle. Traditional processors generally perform "sequential-based" programs where as the FPGA and CGRA can run certain functions of a program in parallel. Executing operations simultaneously saves time and thus, improves the application's energy efficiency.

The goal of this project is to implement an image processing pipeline on a reconfigurable hardware platform to determine the velocity of a chip cutting from a lathe in real-time. The metal hardness can be directly correlated to the chip's velocity. It is desired to show that the optimization of the image processing algorithms in the Stretch S5 CGRA will provide improved performance with respect to run-time speed, power usage, and a more efficient and effective analysis for this application.

II. Background

Few publications discuss image processing techniques to determine material hardness through nondestructive means. Although the aforementioned analysis is a novel technique, previous research that successfully utilized computational analyses of indentation tests warrants the use of image processing algorithms for this project.

Research and industry applications for measuring material hardness involve an invasive method that indents with a tip of a specific material and geometry under a given load. For example, the Rockwell B hardness test uses a 1/16 inch diameter steel ball. These techniques involve a skillful operator analyzing the indentation to evaluate the hardness. However, with current computing power, image processing techniques can be implemented to provide more accurate and efficient results.

Hsu, et. al.[2] utilizes digital image processing to make strain measurements through a multiple regression analysis (MRA) and total least squares (TLS) optimization. The methods involve integrating a mathematical model to locate the lines of elliptical grooves resulting from the stamp of the material. Fine circular patterns are electrochemically etched into the material and a mechanical press stretches the metal. The shape of the ellipse provides information regarding the behavior and characteristics of the material. Both MRA and TLS processing methods provide efficient and detailed means of measuring the ellipse.

Similarly, Yao and Fang[4] developed an automated algorithm called "Hough fuzzy vertex detection algorithm" (HFVDA) for a Vickers hardness test. The method overcomes effects of surface contamination and provides an accurate detection of the indentation edge lines. HFVDA uses the image pixels to locate the edge points from the indentation. A Hough transform is applied to yield four local maxima. A fuzzy c-means (FCM) algorithm finds the center clusters in the Hough space. The local maxima can then be extracted with a simple search, and the indentation edge lines are those lines that pass through the most data points in the Hough space. Although this algorithm involves a destructive technique, it illustrates the increasing usage of image processing in accurately determining material hardness.

Particle Image Velocimetry (PIV) is a similar technique to the approach taken in this paper. PIV is used in fluids where images of fluids seeded with particles are processed to determine the fluid velocity by means of the particle velocity. As mentioned by Hinsch and Herrmann[1], hardware ranging from photographic to charge coupled device (CCD) recording to holographic PIV has been used. However, little of this work has been extended to solids.

III. Hardware Platform

Stretch provides software-configurable processors for compute-intensive applications using C/C++ programming tools. Optimized algorithms can be created within the C/C++ development environment by coding a specialized C-function that can be parallelized into new instructions that execute in a single cycle. The specialized compiler technology automatically converts selected C functions into programmable logic.

The Integrated Development Environment (IDE) tools suite is a graphical interface consisting of a compiler, debugger, assembler, profiler, linker and editor. A debugger and profiler in the IDE tools suite provide verification and analysis capability. In addition, Stretch's C/C++ compiler programs the processor and automatically configures the Instruction-Set Extension Fabric (ISEF) with application-specific instructions. The ISEF is the hardware accelerator that enables numerous calculations to be performed in parallel.

The Stretch S5530 processor uses a single specialized, high performance RISC processor core, the 300-MHz Xtensa. It has 16- and 24-bit instructions as shown in Figure 1. The core supports a memory managed unit (MMU) with a translation look-aside buffer(TLB) (that translates virtual pages to physical pages).

The processor supports one external memory – a 64-bit, 400-Megabits/second (Mbps) double-data-rate synchronous dynamic random access memory (DDR400 SDRAM). This memory achieves greater bandwidth than the preceding single-data-rate SDRAM by transferring data on the rising and falling edges of the clock signal, i.e. it nearly doubles the transfer rate of the system data bus.

The embedded memory specifications are:

- 256-KB SRAM
- 32-KB Data RAM

- 32-KB Data Cache
- 32-KB Instruction Cache

The S5 development board as shown in Figure 2 has support a 32-bit or 64-bit data Peripheral Component Interconnect (PCI) (66 MHz) or PCI-X (133 MHz) port in order to transfer data efficiently to the processor. There are also a host of other common ports for low-data rate information and control such as

- 4 programmable parallel ports
- 2 Time Division Multiplexed (TDM) ports
- 1 Generic Interface Bus (GIB)
- 2 programmable serial ports
- 2 Universal Asynchronous Receive and Transmit (UART) ports
- 1 General Purpose I/O (GPIO) and Interrupts
- 1 standard test port - JTAG (IEEE 1149.1)

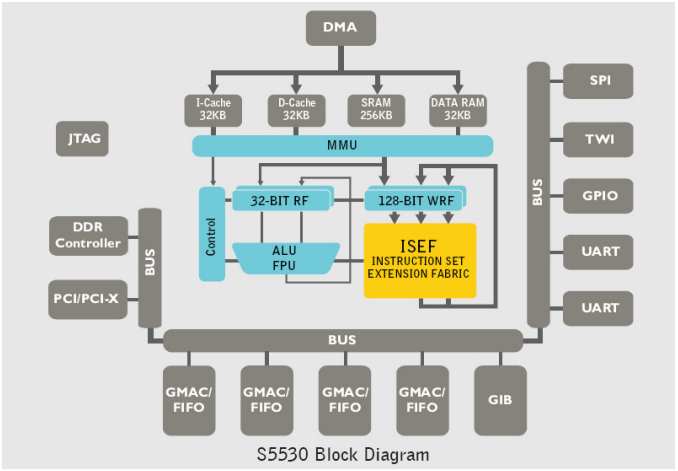

Fig. 1. Stretch S5530 Processor architecture

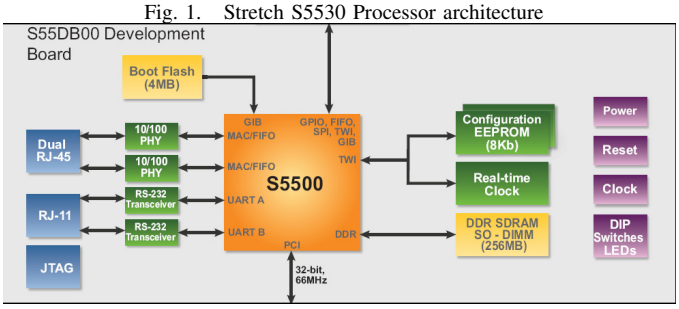

Fig. 2. Stretch S5 Development Board

IV. Algorithms

A. High-Pass Filter

Before any filtration is performed, a black border is added around the image to aid in processing its corner and side elements. Afterwards, a spatial high-pass filter is applied to effectively enhance any sharp edges. To do so, the filter utilizes a k-size window, a k x k pixel array that shifts throughout the image. The difference between the center pixel of each window and the average of all the pixels in that window is performed over the entire image as shown in equation (1).

$$h_{i,j,n} = p_{i+\lfloor \frac{k}{2} \rfloor, j+\lfloor \frac{k}{2} \rfloor, n} - \frac{1}{k^2} \sum_{r=0}^{k-1} \sum_{c=0}^{k-1} p_{i+r,j+c,n} \tag{1}$$

$h_{i,j,n}$ = Result of high pass filter on pixel(i,j) of nth original image (before adding a border)
$p_{i,j,n}$ = Pixel(i,j) of nth bordered image
k = Length and width of high-pass window

For an 8x8 window on an 8-bit image, the outputs can range from -251 to +251, but results will vary if the neighborhood size is changed. A pixel with a value much different than that of its neighbors will have a result that is large in magnitude, while regions of similiar pixel values will have small values following the high-pass filter. Once the frames have been completely processed, the resulting 110x100 images are used as inputs for the interframe differencing.

B. Interframe Differencing

Using the outputs of the high-pass, the interframe differencing omits static values between two subsequent high-pass filter outputs. The algorithm performs a pixel by pixel subtraction, determining the difference between a pixel located in one frame and the corresponding pixel in the next frame as shown in equation (2).

$$d_{i,j,n} = h_{i,j,n} - h_{i,j,n-1} \tag{2}$$

$h_{i,j,n}$ = Result of high pass filter on pixel(i,j) of nth original image
$d_{i,j,n}$ = Interframe difference output for pixel(i,j) between the (n-1)th and nth high-pass filtered image

Resulting values within the arrays have a minimum of -502 or a maximum +502 and the function's output are inputs for the cross-correlation function.

C. Cross-correlation

Finally, two subsequent results from the interframe differencing are used as inputs for the cross-correlation. This function determines where a segment of an image is located in the following frame. The resulting distance, along with a known time, can determine the chip velocity. Using a 58x40 template from one frame, we can shift that template and perform a dot product on the overlapped area in the second image, as equation (3) demonstrates.

$$g_{a,b} = \sum_{\substack{r_{m1}=x_m+a \\ r_{m2}=x_t}}^{\substack{x_m+a+t_x \\ x_t+t_x}} \sum_{\substack{c_{m1}=y_m+b \\ c_{m2}=y_t}}^{\substack{y_m+b+t_y \\ y_t+t_y}} d_{r_{m1},c_{m1},n-1} * d_{r_{m2},c_{m2},n} \tag{3}$$

x_m, y_m = Upper left corner of moving window's initial position
x_t, y_t = Upper left corner of template window's position
t_x, t_y = Template length and width
$d_{i,j,n}$ = Interframe difference output for pixel(i,j)
$g_{a,b}$ = Dot product between moving window shifted (a, b) from its initial position of (n-1)th interframe difference matrix and constant subset of nth interframe difference matrix

The value is then normalized by the product of m1 and m2, each shown below in equations (4) and (5). m1 corresponds to the sum of the absolute value of the numbers in the shifting template region (58x40) and thus varies with the dot product, while m2 remains a constant, calculated once for the constant template region of the second interframe differenced output.

$$m1_{a,b} = \sum_{r=x_m+a}^{x_m+a+t_x} \sum_{c=y_m+b}^{y_m+b+t_y} |d_{r,c,n-1}| \tag{4}$$

$$m2 = \sum_{r=x_t}^{x_t+t_x} \sum_{c=y_t}^{y_t+t_y} |d_{r,c,n}| \tag{5}$$

x_m, y_m = Upper left corner of moving window's initial position
x_t, y_t = Upper left corner of template window's position
t_x, t_y = Template length and width
$d_{i,j,n}$ = Interframe difference output for pixel(i,j) between the (n-1)th and nth bordered image
$m1_{a,b}$ = Normalizing constant for (n-1)th interframe differenced matrix when shifted (a, b) from its initial position
$m2$ = Normalizing constant for nth interframe differenced matrix

Normalization relates the different cross-correlation results from each shift. This operation is performed 286 times, where the one region shifts horizontally 11 times and vertically 26 times while the other is held constant. The highest result, as determined by equation (6) indicates where the segment from the first frame is located.

$$m_{i,j} = \frac{g_{i,j}}{m1_{i,j} * m2} \tag{6}$$

$g_{i,j}$ = Dot product between moving window shifted (i, j) from its initial position of (n-1)th interframe difference matrix and constant subset of nth interframe difference matrix
$m1_{a,b}$ = Normalizing constant for (n-1)th interframe differenced matrix when shifted (a, b) from its initial position
$m2$ = Normalizing constant for nth interframe differenced matrix
$m_{i,j}$ = Normalized cross-correlation result when moving window has been shifted (i, j) from its original position

V. Optimizations

A. Conversion from Floating Point to Integer

The original floating point algorithm was implemented on the S5 processor to provide a baseline for the improved codes and verify the results of any altered sections. The entire program took a total of 7.826 million cycles to process one frame of data, which is equivalent to a maximum throughput of 38 frames per second (fps). In order to obtain the desired 4000 fps the code was modified to perform integer operations. This was done, because in general computers are able to operate faster on integers. This simple modification reduced the program to 5.532 million cycles. However, the processor could still only handle 54 fps. Therefore, additional alterations to the algorithm and utilization of the ISEF were performed to further increase performance.

B. Division

Because division is considerably slower than multiplication and addition on any processor, reducing the time spent performing the operation would considerably increase throughput. Parallelizing the computations reduces the number of cycles, though it does not reduce the number of arithmetic operations. Unfortunately, the accelerator was only designed to handle division by powers of 2, which was implemented as a bit shift. Any other division operations had to be computed in the Xtensa processor.

An approximate number of cycles for the Xtensa processor per division was determined by iterating through two images of 110x100 pixels, dividing each element in one image by the corresponding element in the other. On average, the division took approximately 28 cycles per call, without considering the time spent accessing memory. Reducing the number of iterations to cover an 11x26 matrix (the number used in the cross-correlation function), increased the number of cycles per call to roughly 51.

Avoiding division operations was thus essential. The high-pass filter was implemented by subtracting the average of the 5x5 pixel window from its center pixel value. However, a 5x5 neighborhood required a division by 25. Using window dimensions that were powers of 2 was advantageous, because the accelerator only accepted division by powers of 2. A 4x4 subset passed too many smaller features, so an 8x8 pixel neighborhood was chosen instead for optimization. Although more data must be summed for each pixel, divisions were reduced to shifts of 6 bits. A bit shift operation took just over 6 cycles per call. However, using an 8x8 neighborhood added some complexities. While a 5x5 window had a center pixel, an 8x8 subset did not. The lower right pixel of the four around the center was chosen arbitrarily for this application. Handling edge pixels was simpler for 5x5 neighborhoods, as black border of size (5-1)/2=2 was added all around the image. To adjust the algorithm properly for an 8x8 window, a four-pixel border was used around the top and left edges, while only three pixels were used around the bottom and right sides. Zeros were inserted for the border, so additional complex algorithms were not required to fill in the additional border pixels.

C. Square Root

Calculating a square root took even more cycles than a division. The square root was originally calculated for each element of the 11x26 cross-correlation region. Running a function that passed in a value from an 11x26 matrix, returned the square root of the input, and incremented the address of the result took approximately 937 cycles per call. The resolution after performing the square root was poor when the result was stored as an integer. For example, taking the square root of the numbers 1-3 would return the same result of 1, while 144-168 would all return the value 12.

To remove the square roots, an alternate correlation method was chosen. Rather than finding the sum of the square of each pixel in a template region, the sum of the absolute value of each pixel in the region was used. Computing an absolute value was much less time consuming. Taking the absolute value still prevented negative and positive pixel values from canceling each other and increasing the overall correlation peak. Using the sum did increase the effects of smaller outputs of the interframe difference filter than when each value was squared. On the other hand, intermediate results of the summation were much smaller than using the sums of squares method, yet the final result was much greater than after the square root was taken in the original method. Hence the dot product was divided by a larger number, and thus reduced the magnitude of correlation. Unlike the range of correlation from [-1, 1] using the sums of squares method, the new data needed to be scaled by 10,000 or more to obtain a range with reasonable resolution for integers. The scale factor was dependent on the cross-correlation template size.

D. Array Allocation

Images are generally viewed as 2-D arrays of pixel values. Initial implementations of the algorithms were written using array indexing. However, the use of pointers was quicker. Additionally, the ISEF required passing arguments as pointers rather than arrays. Reads into the accelerator had to consist of sequential bytes, and any pointer jumps that were not multiples of 16 bytes required reinitialization of the data stream. While operations within the ISEF utilized array indexing, transferring data to the accelerator was a more significant issue. In some cases, such as the transpose algorithm, the savings were enormous. A transpose function written with pointers required about 63,000 cycles on a 110x100 short integer matrix, while array notation

took 131,000 cycles. In other functions, the difference was not nearly as severe. If the array could be traversed sequentially, pointers were much quicker because incrementing a pointer was much less expensive than performing an addition operation with a varying operand. Arrays were implemented as a pointer to the first element. Indexing an element in the array added an offset to the pointer, but each time the index was incremented, the new address had to be recalculated from the first element of the array.

E. Parallel Processing of Image Data

Characteristics of the hardware made certain algorithm adjustments beneficial. The Stretch architecture was designed to accelerate code largely by computing sums and/or multiplies in parallel. Computing column sums and then row sums reduced the number of computations significantly. By using row and column sums, an 8x8 pixel block could be computed in 14 parallel sets of addition (seven additions for each of eight column sums, and then another seven additions for each of eight row sums) instead of 63 sequential additions. Furthermore, by having access to three input registers, the overlap of the 8x8 window of adjacent pixels could be utilized. Each neighborhood shared seven common rows or columns (depending on whether the pixels were adjacent vertically or horizontally). To compute the sum of the neighborhood of adjacent pixels, only one column or row sum had to be dropped and the other added. Hence, the three inputs could be used to send in the row or column to be dropped, the current sums, and the row or column to be added. This process could be utilized in both the high-pass filter and cross-correlation.

VI. Results

	Standard	Parallel	Relative Speed
	Processor Cycles (10^6)	Processor Cycles (10^6)	
High-Pass Filter	1.11	0.121	9.17x
Interframe Differencing	0.102	0.0208	4.90x
Cross-Correlation	3.59	1.57	2.30x
Overall	5.55	1.86	2.97x

TABLE I

Algorithm Optimized vs. Unoptimized Performance gains

The usage of the ISEF to parallelize the pixel operations resulted in the largest performance gains without altering the actual operations of the filters. The results were verified against those of the unoptimized implementations, but the actual code had to be modified significantly to make use of the ISEF accelerator.

A. High-Pass Filter

	Standard	Parallel with Transpose	Parallel without Transpose
Processor Cycles (10^6)	1.11	0.191	0.121
Relative Speed	–	5.81x	9.17x

TABLE II

Performance Gains of Parallelized High-Pass Filtering on a Single Image

The parallelization of the high-pass filter resulted in significant gains, as illustrated in Table II. The high-pass filter was reduced from 1.11 million to 191,000 cycles/frame by operating on eight pixels simultaneously. This function ran 5.81 times faster than the version that did not use the accelerator. However, the improved algorithm contained two transposes of the image matrix that were not optimized. These transposes comprised 109,000 of the 191,000 cycles of the filter. Further attempts at optimizing the transpose resulted insignificant gains. Therefore, the algorithm was redesigned to eliminate the transposes. This reduced the high-pass filter to 121,000 cycles/frame, which was 9.17 times faster than the original code.

B. Interframe Difference

Implementing the interframe differencing in parallel also boosted performance significantly, as shown in Table I. The increase in throughput of only 4.90 times was less than expected for operating on eight pixels at once. This gain was also much less than that obtained by the optimized high-pass filter. However, the operations for the interframe differencing were simple subtractions that could already be performed quickly by the processor. The benefit of performing these subtractions in parallel was partially offset by the costs associated with passing data in memory to and from the wide registers.

	Standard	Parallel	Relative Speed
	Processor Cycles (10^6)	Processor Cycles (10^6)	
Cross-Correlation	3.59	3.92	0.916x
Total	3.71	4.09	0.907x

TABLE III

CROSS-CORRELATION RESULTS INITIAL ALGORITHM OPTIMIZED VS. UNOPTIMIZED PERFORMANCE GAINS

C. Cross-correlation

Compared to the other functions of the image processing algorithms, the cross-correlation function took the most time to execute. The initial optimization of the algorithm did not produce any performance gain over the original integer version, as shown in Table III. The original version of the cross-correlation function took 3.59 million cycles/frame whereas the parallelized version took 4.09 million cycles/frame. The 10% increase of cycles was mainly a result of the reconfiguration of the ISEF resources. The hardware had to be rerouted to find the optimal path for file execution every time the accelerator was accessed during the dot product calculation.

The dot product was split into three parts: calculating pixel by pixel products between the two images, adding the rows together, and summing the final resulting row values to obtain the final dot product. The dot product functions were each called 286 times to fill in the values of the 11x26 match array and each call required the accelerator to be accessed. The multiple shifts of the template window significantly contributed to the overall processing time, in addition to the numerous reconfigurations of the ISEF.

	Standard	Parallel	Relative Speed
	Processor Cycles (10^6)	Processor Cycles (10^6)	
Cross-Correlation	3.59	1.50	2.39x
Total	3.71	1.59	2.33x

TABLE IV

CROSS-CORRELATION RESULTS IMPROVED ALGORITHM OPTIMIZED VS. UNOPTIMIZED PERFORMANCE GAINS

Further improvements were made to the dot product of the cross-correlation algorithm, because of the lack of performance gain. An improved algorithm combined all three parts of the dot product to decrease the number of accesses to the ISEF and external memory. Additionally, the function performed a majority of the multiplications and additions simultaneously. These increases in optimization resulted in an execution time of 1.50 million cycles without any overhead for a single image, as indicated in Table IV. Including overhead, the improved parallelized code ran 2.33 times faster than the unoptimized version.

D. Overall Execution

	Unoptimized Floating Point	Unoptimized Integer
	Relative Speed	Relative Speed
Overall Optimized	4.21x	2.97x

TABLE V

COMPARISON BETWEEN OPTIMIZED FUNCTION TO UNOPTIMIZED DATA TYPES

The parallelized version exhibited a performance gain of 4.21 times compared to the floating point implementation with the entire filtration system operating, as shown in Table V. At this speed, a maximum throughput of 161 fps was obtainable. The optimized algorithm was also approximately three times faster than the unoptimized integer implementation.

	Stretch S5 (300 MHz)	Intel Xeon (3.06 GHz)	Relative Performance
Cycles per Frame (10^6)	1.86	17.75	9.54x
Throughput (fps)	161	172	0.936x
Power Consumption (Watts)	3.2	85	26.6x

TABLE VI

COMPARISON OF THE S5 WITH A LINUX PC

Table VI shows a comparison of the parallelized implementation on the Stretch S5 with the unoptimized integer version running on a Linux PC. The computer had a 3.06 GHz Intel Xeon processor using the GNU C-complier. Overall the PC was able to process 11 more frames in a second, but required 9.54 times the number of cycles to process an image than the S5. Even the unoptimized program on the S5 consumed only 5.55 million cycles/frame, which was still over three times less than 17.75 million of the PC. The extra cycles spent on the PC were a result of the overhead associated with running an operating system and the architecture of the chip.

E. Memory

The use of the S5's multiple memory banks was also investigated to improve the efficiency of the program. All variables were stored in the external DDR RAM by default. The DDR RAM offered user-replaceable memory that could be expanded beyond 1GB. This bank offered the largest amount of storage, but it had the greatest latency. The processor could only access this memory at certain times to read and write data. If the memory is unavailable, the processor might waste more than 10 cycles waiting for the DDR RAM [3]. To alleviate this bottleneck, the S5 included zero-latency data RAM built into the computer chip that is always available. However, only 32 KB of data could be stored in this bank, which was only large enough to hold one of the 16-bit image arrays.

A significant portion of all of the algorithms was spent waiting for the memory to become accessible, for example: 44.1% of the processing time was spent copying data to memory in the parallelized high-pass filter that employed transposes. The large amount of time devoted to memory was due largely to high number of pixels per frame and the numerous intermediate steps in the filtering routine that required temporary storage. Therefore, it was deemed necessary to make use of the high-performance data RAM.

Initial investigations using the data RAM to store intermediate outputs were very promising. The optimized high-pass filter operated 2.06 times faster by simply storing two of its intermediate outputs in data RAM. However, the greatest drawback associated with using data RAM was the requirement that all of its variables had to be declared as global attributes. This limitation meant that the variables could not be cleared or resized as the program progressed. Hence, it was impossible to store subsequent two-dimensional arrays, because of the varying output image sizes of the filters.

	Optimized using 2-D array	Optimized using 1-D array	Relative Performance
	Processor Cycles (10^6)	Processor Cycles (10^6)	
High-Pass Filter and Interframe Differencing	0.347	0.252	1.38x

TABLE VII
RESULTS USING DATA RAM IN MULTIPLE LOCATIONS

Storing only one intermediate output in data RAM severely limited its benefits. Therefore, a one-dimensional array large enough to store every pixel in the biggest image was created. Using this style of array did not require reprogramming any sections of the program, because pointer notation was used instead of array indexing throughout the code. The memory allotted for the array was able to store any image without regard to its dimensions. Moreover, it enabled the data RAM to be used throughout all of the filters. Table VII illustrates the benefits of reusing the 1-D array in the high-pass filter and interframe differencing. The parallelized code using the 2-D storage required 347,000 cycles including overhead and memory copying. Simply switching to the 1-D implementation eliminated 27.4% of the cycles required to complete the parallelized code.

The Direct Memory Access capabilities of the Stretch processor offer advantages over the typical PC processor. While many PC processors offer optimized internal memory load and store functionality, the I/O features are much slower. Thus, while Stretch's internal memory speeds may not be remarkable, using an accelerator such as Stretch as an initial data processor to retrieve data and sending compressed versions of the data to a PC can be quite beneficial.

F. Data Type Constraints

One limitation inherent in the C construct of the Stretch language and Xtensa architecture is limited flexibility in data types. While an FPGA can be written with variables of any size, the Stretch processor is primarily limited to multiples of 8 bits. Variables of any size up to 4096 bits can be used within the accelerator, but the main RISC processor passes bytes. Although pixels are represented as 8 bits in the original images, output of the high-pass filter can be from -251 to 251. In practice, the output will rarely reach this magnitude. If the theoretical limit is to be handled, an additional bit must be used to represent the new output. Unfortunately, C then requires the use of a short integer with lengths 16 bits, not 9 bits. The additional 7 bits are wasted. Interframe differencing is similarly limited to 10 bit numbers and in several places throughout the cross-correlation 32-bit integers must be used to prevent overflow. Not only is memory wasted, but fewer pixels can be passed for parallelization. However, FPGAs can parallelize operations as well, and with appropriately sized data buses and gates.

G. Development Experience

Developing the algorithms for the S5 was a relatively simple process. The results in this paper were generated in approximately six weeks by two electrical and one mechanical engineer with no prior knowledge of the Stretch hardware and limited experience in C/C++. This development timeframe was considerably shorter than the time that it would have taken to develop the algorithms on a FPGA.

The portions of the programs not using the ISEF were easily written using the standard C language. Those that did make use of the ISEF took significantly longer to create. The algorithm had to be studied to determine which operations could be done in parallel and the most efficient way to perform these calculations. The greatest difficulties encountered were associated with

transferring data to the wide registers so that it was available to the ISEF and retrieving that data after it had been processed. These transfers were performed with commands specific to the Stretch processor.

Profiling the program was done within the Stretch IDE. This program organized the functions and headers, compiled the programs, and interfaced with the Stretch S5. In addition, the IDE provided cycle-accurate simulations of the program's execution and debugging features to profile and verify the operation of the algorithms. Computational bottlenecks in each section of the program were identified using the graphical profiler. This was very important in the development process because it pointed out troublesome operations such as transposes and square roots.

H. Hardware Limitations

Despite the aforementioned gains made possible by the S5, the performance of the program was greatly hindered by hardware limitations of the system. The aforementioned difficulties associated with memory access slowed the development of the algorithms in addition to the throughput of the filters. Access to the wide-register memory for the ISEF was also problematic. Only being able to read 128 bits per cycle from a wide register meant that only eight of the 16-bit pixels could be operated on in parallel, because most of the filters required corresponding pixels from subsequent images. The ISEF could perform many more operations in parallel if it was able to load more pixels simultaneously. Also, the ISEF could only write two 8-pixel outputs to memory, which limited the performance of the match filter, because it required three intermediate outputs.

These bits could only be read in from sequential memory addresses. Reading in data in this fashion was well-suited for performing operations on rows of the image matrix, but it was impossible to load a column vector of pixels, because their addresses in memory were separated by the number of columns in each row. The input or output stream of the wide-register had to be reinitialized in order to move to a different memory address. Doing so created additional instructions and operations on the pointers to the memory addresses. Unfortunately, multiple stages of the high-pass filter and the cross-correlation function required operations on columns. Therefore, transposes of the matrices were necessary so that row operations could be performed instead. These transposes were not well-suited for parallelization and needed additional memory access. Two transposes made up 57% of the high-pass filter processing time. In order to avoid this bottleneck, the transposes were not performed and these stages of the high-pass filter did not make use of the ISEF at all. The high-pass filter could have been much more efficient if the ISEF was designed to handle column operations or transposes.

VII. Future

The ISEF has 4096 bits of internal storage. To use these bits, static variables must be declared. Utilizing this memory can potentially increase performance by allowing more data loads into the wide registers for parallelization. Currently, functions within the high-pass filter and cross-correlation filter accumulate sums. These functions are implemented by passing the current sum and data to be added into the wide registers, and overwriting the original sum with the new value. However, declaring static variables uses a substantial section of processor resources within the accelerator, leaving less for rerouting datapaths in hardware. Using static variables can provide more wide register inputs, but less ISEF resources can lead to an undesirable result of a throughput decrease. Another limitation in using static variables is the difficulty in retrieving these data values from the ISEF, as static variables are reinitialized each time the ISEF is reconfigured. This may limit the output that can be passed through two wide registers, without using advanced data retrieval techniques. Thus, implementing this modification requires more understanding of the tradeoff.

Another improvement to consider is performing certain functions at the same time. In the cross-correlation filter, the summation of the templates can be executed at the same time as the dot product, because neither segment depends on the other. The summation of the template window can also be done in parallel, but since this is not a moving window, the increase in throughput may not be significant. Additional gain may be realized by combining these operations because the ISEF need not be reconfigured.

During a more recent development, Stretch Inc. has developed an S6 processor. This processor provides 64KB of data RAM, and the accelerator contains 64KB of embedded ISEF RAM (IRAM). IRAM can be used for coefficients, intermediate results, or look-up tables. Thus, implementing static variables will not be necessary on the S6 processor. The IRAM can also be loaded without any processor intervention, providing greater throughput.

Another idea to speed performance would be to combine parts of filters. Once the high-pass filtered image has filtered the second image, the interframe differencing can be performed without exiting the accelerator. This can reduce any unnecessary ISEF configuration time, as well as eliminate the need to send the data an extra time to the accelerator.

Further investigations into the cross-correlationŠs template regions can provide insight for an optimal window size, offering similar results for less computational overhead.

The aforementioned improvements can increase throughput of the image processing algorithms and potentially lower power consumption.

VIII. Conclusion

The C language format used by the Stretch compiler provides a development environment with a gentler learning curve than that of FPGAs. An unoptimized program can be written quickly with minimal programming experience, though an understanding of pointers would be beneficial for writing code. Sections of the code that require more processor cycles can then be optimized. Although all aspects of the program can be optimized to deeply explore the processor and its capabilities, large gains could be realized more quickly by focusing mostly on the cross-correlation filter.

Overhead does significantly reduce performance gain. For the cross-correlation calculation function, configuring the ISEF multiple times in an initial version of the code actually made the performance worse than without accelerating the code. Even after the code was rewritten, memory storage rates and data sizes reducing the number of operations handled in parallel prevented performance gains from going above 3x.

While the gains were not nearly as high as desired, the speed of the algorithm in the Stretch simulator did nearly reach that of the Intel processor. With further work, this could likely be surpassed. Even now, the DMA capabilities and configurable nature make this a viable alternative to a PC. Working in conjunction with a PC, even better results may be attainable.

References

[1] Klaus D. Hinsch and Sven F. Herrmann. Holographic particle image velocimetry. *Measurement Science and Technology, vol 15, issue 4*, April 2004.

[2] Q.-C. Hsu. Comparison of different analysis models to measure plastic strains on sheet metal forming parts by digital image processing. *International Journal of machine Tools & Manufacture,vol. 43, pp. 515-521*, 2003.

[3] Stretch Inc. Stetrch s5 processor. *http://www.stretchinc.com*, 1997.

[4] L. Yao and C. Fang. A hardness measuring method based on hough fuzzy vertex detection algorithm. *IEEE Trans. Ind. Electron., vol. 53, no. 3, pp. 950-962*, June 2006.